CS: a MuPAD Package for Counting and Randomly Generating Combinatorial Structures

Alain Denise, Isabelle Dutour, Paul Zimmermann

CS: a MuPAD Package for Counting and Randomly Generating Combinatorial Structures

Alain Denise
LRI, Université Paris-Sud
Alain.Denise@lri.fr

Isabelle Dutour
LORIA, INRIA Lorraine
Isabelle.Dutour@loria.fr

Paul Zimmermann
LORIA, INRIA Lorraine
Paul.Zimmermann@loria.fr

April 10, 1998

Abstract

We present a new computer algebra package which permits to count and to generate combinatorial structures of various types, provided that these structures can be described by a specification, as defined in [7].

Résumé

Nous présentons un nouveau module de calcul formel dédié au dénombrement et à la génération aléatoire uniforme de structures combinatoires décomposables.

1 What is CS ?

CS is a computer algebra package devoted to the handling of combinatorial structures. Its main features are the following: given a combinatorial specification of a class of *decomposable* structures (in the sense of [7]), CS is able to count and uniformly draw at random the structures of any given size n. It can also give some properties of the associated generating series, like recurrences and differential equations.

A *specification* of a class of combinatorial structures, as defined in [7], is a set of productions made from basic objects (atoms) (**Epsilon** and **Z** of size 0 and 1 respectively) and from constructions (**Union** for disjoint unions, **Prod** for products, **Sequence** for sequences, **Set** for sets (labelled case) or multisets (unlabelled case)). A class of structures is called *decomposable* when it admits a specification with a finite number of basic objects and constructions. For example, binary trees, ternary trees, permutations, surjections, functional graphs, integer or set partitions, hierarchies, are all decomposable structures. An example of non-decomposable structures is the class of general graphs, as cutting an edge in a graph does not usually divide it into two subgraphs.

CS, like COMBSTRUCT (previously known as Gaïa [11]), another package developed by one of the authors, is mainly based on the theory developed in [7] and [6]. But, in addition to the fact that CS is a MuPAD package while COMBSTRUCT is a Maple package, CS presents some more advanced features: at the present time, CS offers the same functionalities than COMBSTRUCT offers concerning unlabelled structures (except for cycles and some conditions on cardinality of sets), plus the following:

- Given any decomposable class, CS is able to automatically generate the source of a C program for almost uniform random generation. Then this program can be compiled and used totally independently from MuPAD. You are then able to get the approximate counting or draw at random one or several of its elements of desired size, much more efficiently than with a Maple or a MuPAD procedure. This is the first implementation of one of the algorithms given in [4].
- When the generating series of a decomposable class is holonomic, CS is able to compute, using Gröbner basis calculation and Gaussian elimination, a linear recurrence that its coefficients satisfy. This improves the complexity of counting and enables the generation of larger structures.

At the present time, CS only deals with unlabelled structures. The CS package is currently being developed and we plan to add the labelled case and several other new features; Section 6 describes some of them.

2 How to install CS on your computer

First, you must have installed the MuPAD computer algebra system on your computer. This software is distributed for free (but not in public domain: a registration is required). It can be downloaded at the following address:

```
http://www.mupad.de
```

MuPAD is available on several computer systems and the way to install it on each system is precisely described.

Once you have MuPAD on your computer, you can download the CS package (`cs.mu`, one ASCII file) at the address:

```
http://www.loria.fr/~dutour/CS
```

Now you just have to run MuPAD and to read CS by typing, after the prompt "`>>`",

```
read("cs.mu"):
```

3 Counting and Generating

Here is the MuPAD instruction which creates the specification of plane trees:

```
>> specPT:={T=Prod(Z,Sequence(T))};

                    {T = Prod(Z, Sequence(T))}
```

In other words, a plane tree is a product of a root vertex (the atom `Z`) and a sequence (possibly empty) of plane subtrees. To count the number of plane trees of size, say 100, use the `count` procedure of the CS package:

```
>> cs::count([T,specPT],size=100);

  22750883079422934966181954039568885395604168260154104734 0
```

We can verify that the first coefficients of the generating series correspond to the first Catalan numbers:

```
>> cs::count([T,specPT],size=i)$i=1..10;

          1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862
```

Now let us draw a random plane tree with 10 vertices:

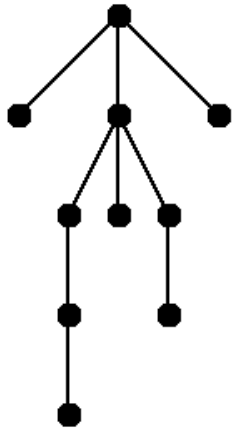
```
>> cs::draw([T,specPT],size=10);

Prod(Z, Sequence(Prod(Z, Epsilon), Prod(Z, Sequence(

   Prod(Z, Sequence(Prod(Z, Sequence(Prod(Z, Epsilon))))), Prod(Z, Epsilon)

   , Prod(Z, Sequence(Prod(Z, Epsilon))))), Prod(Z, Epsilon)))
```

This description corresponds to the following tree:



Remark: Such a figure can be drawn almost directly from the output of the `draw` procedure by using CalICo [3] or CGraph [2], two programs dedicated to the manipulation and visualization of combinatorial structures. This figure was drawn with CalICo.

Here is another example of a specification, using an argument of cardinality; `P` represents the integer partitions and `N` the integers greater or equal to 1:

```
>> specPART:={P=Set(N),N=Sequence(Z,card>=1)};

                {N = Sequence(Z, 1 <= card), P = Set(N)}

>> cs::draw([P,specPART],size=5);

Set(Sequence(Z), Sequence(Z), Sequence(Z), Prod(Sequence(Z, Z))
```

The `count` and `draw` procedures both call the `compile` procedure. This last one produces, given the specification, the recurrences which will be used in order to count and generate the

structures. The variable `HOLONOMIC`, when switched to the value `TRUE`, tells CS to test, when compiling a class of structures, if its generating function is holonomic or not. If yes, CS finds a simple linear recurrence to count the structures, so that the counting runs more quickly, i.e. in linear instead of quadratic number of operations. (The default value of `HOLONOMIC` is `FALSE`). To illustrate this, compare the time needed to count the number of Motzkin trees of size 100, when compiling with `HOLONOMIC` equal to `FALSE` or `TRUE` (on a Sun Sparc 10, CPU time in ms):

```
>> specMT:={M=Union(Z,Prod(Z,M),Prod(Z,M,M))};

                {M = Union(Z, Prod(Z, M), Prod(Z, M, M))}


>> cs::HOLONOMIC:=FALSE:
>> st:=time(): cs::compile(specMT): time()-st;

                                    6330


>> st:=time(): cs::count([M,specMT],size=100); time()-st;

              249478578991224378680142561460010030467811580

                                    36160


>> cs::HOLONOMIC:=TRUE:
>> st:=time(): cs::compile(specMT): time()-st;

                                    19470


>> st:=time(): cs::count([M,specMT],size=100); time()-st;

              249478578991224378680142561460010030467811580

                                    1160
```

In the second case, the call to `compile` is longer, due to the computation using Gröbner basis and Gaussian elimination; but once finished, the counting and random generation are much more efficient. The next section will detail what happens exactly when compiling with `HOLONOMIC=TRUE`.

4 Equations and Recurrences

The CS package provides some procedures in order to handle generating series. They implement (in MuPAD) some functionalities of the GFUN Maple package [10]. These procedures are used in `compile` (see the details below) but can also be called independently.

The procedure `speciftoalgeq` returns an algebraic equation that the generating series associated to the given specification satisfies, provided that such an equation exists:

```
>> eq:=cs::speciftoalgeq(specMT,M);
                                          2
                          z - M + M z + M  z
```

(At first glance, this does not look like an equation, but this is to be read as $z - M + Mz + M^2 z = 0$.)

The procedure `algeqtodiffeq` converts this equation into a differential equation, while `diffeqtorec` gives the recurrence on the coefficients involved by the differential equation:

```
>> deq:=cs::algeqtodiffeq(eq,M(z));
                                          2              3
      2 z - M(z) + z M(z) - z D(M)(z) + 2 z  D(M)(z) + 3 z  D(M)(z)

>> r:=cs::diffeqtorec(deq, M(z), v(n));

    v(n - 1) - v(n) - n v(n) + 2 (n - 1) v(n - 1) + 3 (n - 2) v(n - 2)
```

We can resume all these steps in one unique call:

```
>> r:=cs::speciftorec(specMT,M,v(n));

    v(n - 1) - v(n) - n v(n) + 2 (n - 1) v(n - 1) + 3 (n - 2) v(n - 2)
```

The procedure `rectoproc` outputs a MuPAD procedure that gives the n-th term of the linear recurrence given as input. The `cs::LIM` variable is set to the rank from which the recurrence is valid. The user has to initialize himself the first values up to this rank.

```
>> m:=cs::rectoproc(r, v(n));
proc(n)
  name m;
  local nmax, j, g, ff;
begin
  nmax:=max(op(map([op(op(level(procname), 5))], op, 1)));
  for j from nmax + 1 to n do
    ff:=level(procname);
    g:=procname(j);
    evalassign(g, (j*(-1) + (-1))^(-1)*(ff(j + (-1))*(j*2 + (-1)) + ff(j +\
 (-2))*(j*3 + (-6)))*(-1), 1)
  end_for
end_proc

>> cs::LIM;
                                       2

>> m(0):=0: m(1):=1: m(2):=1:
>> m(100);
              24947857899122437868014256146001003046781158O
```

All these procedures are called by the `compile` function when the variable `HOLONOMIC` is equal to `TRUE`. You can take again the last example of the previous section and, before compiling with `HOLONOMIC=TRUE`, call the MuPAD function `setuserinfo` in order to see the operations involved by the calculation of the linear recurrences:

```
>> cs::HOLONOMIC:=TRUE:
>> setuserinfo(Any,1):
>> cs::compile(specMT);
...
standard form is: , {T1 = Prod(Z, M), T3 = ProdPrd(M, M), T2 = Prod(Z, T3)\
, Z = Atom(Z), M = Union(Z, T1, T2)}
...
polynomials are: , [T1*(-1) + M*z, T3*(-1) + M^2, T2*(-1) + z*T3, M*(-1) +\
 z + T1 + T2]
auto reduces list of 4 polynomials
groebner basis is: , [M^2 + M*z^(-1)*(z + (-1)) + 1, T1 + M*z*(-1), T3 + M\
*z^(-1)*(z + (-1)) + 1, z + T2 + M*(z + (-1))]
...
perform (ordinary) Gaussian elimination
algeq is: , T3*(-1) + z*T3*2 + z^2 + z^2*T3 + z^2*T3^2
perform (ordinary) Gaussian elimination
diffeq is: , T3(z)*2 + z*T3(z)*(-4) + z*D(T3)(z) + z^2*(-4) + z^2*T3(z)*(-\
2) + z^2*D(T3)(z)*(-3) + z^3*D(T3)(z)*(-1) + z^4*D(T3)(z)*3
recurrence for , T3,  is: , u(n)*2 + n*u(n) + u(n + (-1))*(-1) + u(n + (-3\
))*(-9) + n*u(n + (-1))*(-3) + n*u(n + (-2))*(-1) + n*u(n + (-3))*3
...
                                    TRUE
```

(In the computation above, some parts of the outpout are missing, because the full output would be too long for this presentation ; we just kept some significative extracts.)

What happens with `cs::HOLONOMIC:=TRUE` is the following. Firstly, CS translates the standard form of the user specification into a set of polynomial equations for the corresponding generating functions. Here, the nonterminals are M, T_1, T_2, T_3 and the equations are

$$zM - T_1 = 0, M^2 - T_3 = 0, zT_3 - T_2 = 0, z + T_1 + T_2 - M = 0.$$

Secondly a Gröbner basis for this system of polynomials is computed, with coefficients which are rational functions in z. This basis is here:

$$M^2 + \frac{z-1}{z}M + 1, T_1 - zM, T_3 + \frac{z-1}{z}M + 1, z + T_2 + (z-1)M.$$

Then for each nonterminal T — only the case of T_3 is shown above — one reduces $1, T, T^2, \ldots$ with respect to the Gröbner basis, until a linear dependency is found, with coefficients being rational functions in z. Such a dependency necessarily exists because the number of possible monomials arising in the reductions is finite, as the corresponding ideal is zero-dimensional. The dependency found for T_3 is

$$z^2T_3^2 + (z^2 + 2z - 1)T_3 + z^2 = 0.$$

It follows from the theory of holonomic functions that every algebraic function satisfies a linear differential equation with polynomial coefficients. Such a differential equation for T_3, which can be computed by the `algeqtodiffeq` function, is:

$$(3z^4 - z^3 - 3z^2 + z)T_3'(z) + (-2z^2 - 4z + 2)T_3(z) - 4z^2 = 0.$$

This differential equation leads in turn to a linear recurrence for the Taylor coefficients of $T_3(z)$:

$$(n+2)u_n - (3n+1)u_{n-1} - nu_{n-2} + (3n-9)u_{n-3} = 0.$$

This recurrence, together with the initial coefficients, enables one to compute all coefficients up to order n in $O(n)$ operations.

5 Generation of C code

The Schröder trees are taken as example in this section. Each internal node of theses trees has at least two subtrees. Their enumeration according to their number of leaves leads to the Schröder numbers. Here all the nodes are counted:

```
>> specST:={S=Union(Z,Prod(Z,Sequence(S,card>=2)))}:
```

The `compile` procedure allows additional arguments in order to create the source file of a C program that can generate more quickly one or several structures of a given size. This program can be compiled and run either from or out of MuPAD. The following example shows how to process the first possibility.

```
>> cs::compile(specST,target=C,file="SchroederTree.c",main=S);
```

```
                                TRUE
```

```
>> system("cc -Dsun4 SchroederTree.c -lm; a.out 5");
Prod(Z,Sequence(Z,Z,Z,Z))
```

```
                                1
```

The binary file `a.out` can take two arguments. The first is the size of the structure (n) and the second is the number of structures you want to generate. If this second argument is not present, the program generates only one structure, and if it is equal to 0, it gives the approximate number of structures of size n:

```
>> system("a.out 1000 0");
5.1077756867799034*10^471..5.1077756867845379*10^471
```

```
                                1
```

In fact, the result consists in an interval; we can guarantee that the exact number of structures lies in this interval, because the program does respect the IEEE specifications concerning floating-point calculations. This can be easily verified for our example:

```
>> cs::count([S,specST],size=1000);

5107775686782111131410747188352048764645996293961521213214056228358653155\0
8571253191797215185585333814842888485404070873849912292730731099522320697\2
4664772659627384315546489403128555828619079579639369516558153043429253773\0
5936175175325800770283228011328211760136485932236365039827447954873362478\2
4501876852799975209595490917320598561734826212953344100857067317657225537\5
4712168933988535120958880164912956897098711561506379836847710447809817059\9
5549956379942013897484633052

>> length(%);
                                472
```

The C program allows to manipulate very large numbers (see above). Normally, it can be compiled on all machines. Presently, the supported architectures are SunOs (-Dsunos), Solaris (-Dsun4), DecAlpha (-Dalpha), IRIX (-DIRIX64), HP (-Dhp700).

The following table compares the CPU time (in seconds on an IRIX machine) needed for all computations. The two columns "count" give the average time required for the preprocessing, without or with linear recurrences (quadratic and linear respectively), while the column "draw" gives the average time for one random generation (over 100 generations, except for the size 1000000 for which just one generation was processed), which is quasi-linear.

n	count (without rec.)	count (with rec.)	draw
100	0.037	0.007	0.002
1000	4.158	0.036	0.027
10000	499.210	0.358	0.346
100000	–	4.054	4.364
1000000	–	45.219	57.352

Remark: in the current version of CS, the C program only performs *almost uniform* random generation, because the coefficients used in the generation procedures are computed in floating-point arithmetic. (However, since the procedures are numerically stable, the rounding errors involved by floating-point arithmetic do not strongly affect the quasi-uniformity of generation; for details, see [4]). A future version of CS will allow to produce exactly uniform generators in C language (see Section 6).

6 Future Features

The CS package is presently at the beginning of its development (the current version number is 1.0). Our aim is to add, within the next few months, several new features. Here are some of them.

- A future version of CS will handle labelled structures as well as unlabelled ones, and the cycle constructor.
- When a specification is not context-free, CS can not compute linear recurrences to count the objects (in linear number of operations), and the countings runs in quadratic number of operations (due to the products). In this case, we want to use the fast lazy

multiplication algorithm of formal power series designed by Joris van der Hoeven [9]. We are implementing it by using Karatsuba's algorithm as sub-algorithm, and we obtain an arithmetic complexity — i.e. the number of operations on the series coefficients — of $O(n^{\log 3/\log 2})$, with respect to $O(n^2)$ with the classical algorithm.

- One of the authors has developed a Maple package, named qAlGO, mainly based on the notion of object grammars [5], and that leads to enumeration and random generation of objects according to non algebraic parameters. The packages qAlGO and COMBSTRUCT complement each other. We plan to integrate also the qAlGO technics in CS.

- Recently, two of the authors investigated a method, based on a strategy of "lazy evaluation", which fairly improves the average complexity of the exactly uniform random generation of decomposable structures [4]. We plan to use this method in CS in order to get more efficient generation procedures.

- In the current version of CS, the C programs written by the `draw` procedure only perform almost uniform random generation. We intend to use a multiprecision mathematical library, like PARI [1] or GMP [8], so that CS can write true random generators in C language.

References

[1] C. Batut, D. Bernardi, H. Cohen, and M. Olivier. *User's Guide to PARI-GP*, January 1995. URL `ftp://megrez.math.u-bordeaux.fr/pub/pari`.

[2] F. Bertault. *Génération et tracé de structures décomposables.* PhD thesis, Université Henri Poincaré Nancy 1, September 1997.

[3] M. Delest, J.M. Fédou, G. Melançon, and N. Rouillon. Computation and images in combinatorics. In RIACA Amsterdam, editor, *HISC book.* Springer Verlag, To appear.

[4] A. Denise and P. Zimmermann. Uniform random generation of decomposable structures using floating-point arithmetic. Technical Report 3242, INRIA, 1997. To appear in Theoretical Computer Science.

[5] I. Dutour and J.M. Fédou. Object grammars and random generation. Technical Report 1165-97, LaBRI, Université Bordeaux I, 1997. To appear in Discrete Mathematics and Theoretical Computer Science.

[6] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus of random generation: Unlabelled structures. In preparation.

[7] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132:1–35, 1994.

[8] T. Granlund. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 2.0.2 edition, June 1996.

[9] J. van der Hoeven. Lazy multiplication of formal power series. In W. W. Küchlin, editor, *Proc. ISSAC'97*, pages 17–20, Maui, Hawaii, July 1997.

[10] B. Salvy and P. Zimmermann. Gfun: A Maple package for the manipulation of generating and holonomic functions in one variable. *ACM Transactions on Mathematical Software*, 20(2):163–177, June 1994.

[11] P. Zimmermann. Gaïa: a package for the random generation of combinatorial structures. *MapleTech*, 1(1):38–46, 1994.